Конспект лекции
Модульное тестирование на базе JUnit5. Основы тестирования

Цель и задачи лекции

Цель – научиться писать модульные тесты на базе JUnit5.

Задачи:

1. Ознакомиться с понятием модульного тестирования
2. Узнать, как использовать JUnit5

План занятия

1. Основы тестирования
2. Обзор JUnit5

Основы тестирования

Задачи и цели модульного тестирования

Каждая сложная программная система состоит из отдельных частей - модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе всей системы, необходимо вначале протестировать каждый модуль системы по отдельности. В случае возникновения проблем при тестировании системы в целом это позволяет проще выявить модули, вызвавшие проблему, и устранить соответствующие дефекты в них. Такое тестирование модулей по отдельности получило называние модульного тестирования (unit testing).

Для каждого модуля, подвергаемого тестированию, разрабатывается тестовое окружение, включающее в себя драйвер и заглушки, готовятся тест-требования и тест-планы, описывающие конкретные тестовые примеры.

Основная цель модульного тестирования - удостовериться в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы.

При этом в ходе модульного тестирования решаются следующие основные задачи:

- Поиск и документирование несоответствий требованиям
- Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия
- Поддержка рефакторинга модулей
- Поддержка устранения дефектов и отладки

Первая задача - классическая задача тестирования, включающая в себя не только разработку тестового окружения и тестовых примеров, но и выполнение тестов, протоколирование результатов выполнения, составление отчетов о проблемах.

Вторая задача больше свойственна "легким" методологиям типа XP, где применяется принцип тестирования перед разработкой (Test-driven development), при котором основным источником требований для программного модуля является тест, написанный до реализации самого модуля. Однако, даже при классической схеме тестирования модульные тесты могут выявить проблемы в дизайне системы и нелогичные или запутанные механизмы работы с модулем.

Третья задача связана с поддержкой процесса изменения системы. Достаточно часто в ходе разработки требуется проводить рефакторинг модулей или их групп - оптимизацию или полную переделку программного кода с целью повышения его сопровождаемости, скорости работы или надежности. Модульные тесты при этом являются мощным инструментом для проверки того, что новый вариант программного кода выполняет те же функции, что и старый.

Последняя, четвертая, задача сопряжена с обратной связью, которую получают разработчики от тестировщиков в виде отчетов о проблемах. Подробные отчеты о проблемах, составленные на этапе модульного тестирования, позволяют локализовать и устранить многие дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

В силу того, что модули, подвергаемые тестированию, обычно невелики по размеру, модульное тестирование считается наиболее простым (хотя и достаточно трудоемким) этапом тестирования системы. Однако, несмотря на внешнюю простоту, с модульным тестированием сопряжены две проблемы.

Первая из них связана с тем, что не существует единых принципов определения того, что в точности является отдельным модулем.

Вторая заключается в различиях в трактовке самого понятия модульного тестирования - понимается ли под ним обособленное тестирование модуля, работа которого поддерживается только тестовым окружением, или речь идет о проверке корректности работы модуля в составе уже разработанной системы. В последнее время термин "модульное тестирование" чаще используется во втором смысле, хотя в этом случае речь скорее идет об интеграционном тестировании.

Понятие модуля и его границ. Тестирование классов

Традиционное определение модуля с точки зрения его тестирования: "модуль - это компонент минимального размера, который может быть независимо протестирован в ходе верификации программной системы". В реальности часто возникают проблемы с тем, что считать модулем. Существует несколько подходов к данному вопросу:

- модуль - это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований;
- модуль - это программный модуль, т.е. минимальный компилируемый элемент программной системы;
- модуль - это задача в списке задач проекта (с точки зрения его менеджера);
- модуль - это участок кода, который может уместиться на одном экране или одном листе бумаги;
- модуль - это один класс или их множество с единым интерфейсом.

Обычно за тестируемый модуль принимается либо программный модуль (единица компиляции) в случае, если система разрабатывается на процедурном языке программирования, или класс, если система разрабатывается на объектно-ориентированном языке.

В случае систем, написанных на процедурных языках, процесс тестирования модуля происходит так, как это было рассмотрено в предыдущих лекциях: для каждого модуля разрабатывается тестовый драйвер, вызывающий функции модуля и собирающий результаты их работы, и набор заглушек - они имитируют поведение функций, которые содержатся в других модулях, не попадающих под тестирование данного модуля. При

тестировании объектно-ориентированных систем существует ряд особенностей, прежде всего вызванных инкапсуляцией данных и методов в классах.

В случае объектно-ориентированных систем более мелкое деление классов и использование отдельных методов в качестве тестируемых модулей нецелесообразно, поскольку для тестирования каждого метода потребуется разработка тестового окружения, сравнимого по сложности с уже написанным программным кодом класса. Кроме того, декомпозиция класса нарушает принцип инкапсуляции, согласно которому объекты каждого класса должны вести себя как единое целое с точки зрения других объектов.

Процесс тестирования классов как модулей иногда называют компонентным тестированием. В ходе такого тестирования проверяется взаимодействие методов внутри класса и правильность доступа методов к внутренним данным класса. Здесь возможно обнаружение не только стандартных дефектов, связанных с выходами за границы диапазона или неверно реализованными требованиями, но и специфических дефектов объектно-ориентированного программного обеспечения:

- дефектов инкапсуляции, в результате которых, например, сокрытые данные класса оказываются недоступными для соответствующих публичных методов;
- дефектов наследования, при наличии которых схема наследования блокирует важные данные или методы от классов-потомков;
- дефектов полиморфизма, при которых полиморфное поведение класса оказывается распространенным не на все возможные классы;
- дефектов инстанцирования, при которых во вновь создаваемых объектах класса не устанавливаются корректные значения по умолчанию параметров и внутренних данных класса.

Однако, выбор класса в качестве тестируемого модуля имеет и ряд сопряженных проблем.

Определение степени полноты тестирования класса. В том случае, если в качестве тестируемого модуля выбран класс, не совсем ясно, как определять степень полноты его тестирования. С одной стороны, можно использовать классический критерий полноты покрытия программного кода тестами: если полностью выполнены все структурные элементы всех методов, как публичных, так и скрытых, то тесты можно считать полными.

Однако существует альтернативный подход к тестированию класса, в котором все публичные методы должны предоставлять пользователю данного класса согласованную схему работы - тогда достаточно проверить типичные корректные и некорректные сценарии работы с данным классом. Т.е., например, в классе, объекты которого представляют записи в телефонной книжке, одним из типичных сценариев работы будет "Создать запись -> искать запись и найти ее -> удалить запись -> искать запись вторично и получить сообщение об ошибке".

Различия в этих двух методах напоминают различия между тестированием черного и белого ящиков, но, на самом деле, второй подход отличается от черного ящика тем, что функциональные требования на

систему могут быть составлены на уровне более высоком, чем отдельные классы, и установление адекватности тестовых сценариев требованиям остается на откуп тестировщику.

Протоколирование состояний объектов и их изменений. Некоторые методы класса предназначены не для выдачи информации пользователю, а для изменения внутренних данных объекта класса. Значение внутренних данных объекта определяет его состояние в каждый отдельный момент времени, а вызов методов, изменяющих данные, изменяет и состояние объекта. При тестировании классов необходимо проверять, что класс адекватно реагирует на внешние вызовы в любом из состояний. Однако, зачастую из-за инкапсуляции данных невозможно определить внутреннее состояние класса программными способами внутри драйвера.

В этом случае может помочь составление схемы поведения объекта как конечного автомата с определенным набором состояний. Такая схема может входить в низкоуровневую проектную документацию (например, в составе описания архитектуры системы), а может составляться тестировщиком или разработчиком на основе функциональных требований к системе. В последнем случае для определения всех возможных состояний может потребоваться ручной анализ программного кода и определение его соответствия требованиям. Автоматизированное тестирование в этом случае может лишь определить, по всем ли выявленным состояниям осуществлялись переходы и все ли возможные реакции проверялись.

Тестирование изменений. Как уже упоминалось выше, модульные тесты - мощный инструмент проверки корректности изменений, внесенных в исходный код при рефакторинге. Однако, в результате рефакторинга только одного класса, как правило, не меняется его внешний интерфейс с другими классами (интерфейсы меняются при рефакторинге сразу нескольких классов). В результате обычных эволюционных изменений системы у класса может меняться внешний интерфейс, причем как по формальным (изменяются имена и состав методов, их параметры), так и по функциональным (при сохранении внешнего интерфейса меняется логика работы методов) признакам. Для проведения модульного тестирования класса после таких изменений потребуется изменение драйвера и, возможно, заглушек. Но только модульного тестирования в данном случае недостаточно, необходимо также проводить и интеграционное тестирование данного класса вместе со всеми классами, которые связаны с ним по данным или по управлению.

Вне зависимости от того, на какие модули, подвергаемые тестированию, разбивается система, рекомендуется изложить принципы выделения тестируемых модулей в плане и стратегии тестирования, а также составить на базе структурной схемы архитектуры системы новую структурную схему, на которой нужно отметить все тестируемые модули. Это позволит спрогнозировать состав и сложность драйверов и заглушек, требуемых для модульного тестирования системы. Такая схема также может использоваться позже на этапе модульного тестирования для выделения укрупненных групп модулей, подвергаемых интеграции.

Подходы к проектированию тестового окружения

Вне зависимости от того, какая минимальная единица исходных кодов системы принята за минимальный тестируемый модуль, существует еще одно различие в подходах к модульному тестированию.

Первый подход к модульному тестированию основывается на предположении, что функциональность каждого вновь разработанного модуля должна проверяться в автономном режиме без его интеграции с системой. При таком подходе для каждого вновь разрабатываемого модуля создаются тестовый драйвер и заглушки, с помощью которых выполняется набор тестов. Только после устранения всех дефектов в автономном режиме производится интеграция модуля в систему и проводится тестирование на следующем уровне. Достоинством данного подхода является более простая локализация ошибок в модуле, поскольку при автономном тестировании исключается влияние остальных частей системы, которое может вызывать маскировку дефектов (эффект четного числа ошибок). Основной недостаток данного метода - повышенная трудоемкость написания драйверов и заглушек, поскольку заглушки должны адекватно моделировать поведение системы в различных ситуациях, а драйвер должен не только создавать тестовое окружение, но и имитировать внутреннее состояние системы, в составе которой будет функционировать модуль.

Второй подход построен на предположении, что модуль все равно работает в составе системы и если модули интегрировать в систему по одному, то можно протестировать поведение модуля в составе всей системы. Этот подход свойственен большинству современных "облегченных" методологий разработки, в том числе и XP.

В результате применения такого подхода резко сокращаются трудозатраты на разработку заглушек и драйверов - в роли заглушек выступает уже оттестированная часть системы, а драйвер выполняет только функции передачи и приема данных, не моделируя внутреннее состояние системы.

Тем не менее, при использовании данного метода возрастает сложность написания тестовых примеров - для приведения системы в нужное состояние системы заглушек, как правило, требуется только установить значения тестовых переменных, а для приведения в нужное состояние части реальной системы необходимо выполнить сценарий, приводящий в это состояние. Каждый тестовый пример в этом случае должен содержать такой сценарий.

Кроме того, при этом подходе не всегда удается локализовать ошибки, скрытые внутри модуля, которые могут проявиться при интеграции следующих модулей.

Организация модульного тестирования

Модульное тестирование, с точки зрения тестировщика, - это комплекс работ по выявлению дефектов в тестируемых модулях. В эти работы включается анализ требований, разработка тест-требований и тест-планов,

разработка тестового окружения, выполнение тестов, сбор информации об их прохождении.

Однако, с точки зрения руководителя группы тестирования (или с точки зрения руководителя проекта, если в нем не выделена отдельная группа тестирования), модульное тестирование является более широким понятием. Для того, чтобы процесс модульного тестирования мог функционировать совместно с другими процессами разработки, он должен включать в себя несколько фаз: планирование процесса, разработку тестов, выполнение тестов, сбор статистики, управление отчетами о выявленных дефектах.

Согласно стандарту IEEE 1008 процесс модульного тестирования состоит из трех фаз, в состав которых входит 8 видов деятельности (этапов).

- Фаза планирования тестирования
  - Этап планирования основных подходов к тестированию, ресурсное планирование и календарное планирование
  - Этап определения свойств, подлежащих тестированию
  - Этап уточнения основного плана, сформированного на этапе (1)
- Фаза получения набора тестов
  - Этап разработки набора тестов
  - Этап реализации уточненного плана
- Фаза измерений тестируемого модуля
  - Этап выполнения тестовых процедур
  - Этап определения достаточности тестирования
  - Этап оценки результатов тестирования и тестируемого модуля.

Во время этапа планирования основных подходов в качестве входных данных используется общий план проекта (модульное тестирование, как часть проектных работ, должно укладываться в общий график) и требования к системе (для оценки трудоемкости работ и любого планирования необходимо проводить анализ сложности системы на основании требований к ней).

Основные задачи, решаемые в ходе этапа планирования, включают в себя:

- определение общего подхода к тестированию модулей - определяются риски и на их основе - степень полноты и охвата тестирования системы. Определяются источники входных и выходных данных. Определяются технологии проверки результатов тестирования и форматы записи данных о проведенном тестировании. Описывается внешний интерфейс тестируемых модулей и их информационное окружение;
- определение требований к полноте тестирования - определяется необходимая степень покрытия программного кода различных участков тестируемого модуля, определяется подходы к классам эквивалентности (требуется ли тестирование за границами диапазона);
- определение требований к завершению тестирования - определяются условия, проверка которых позволяет утверждать, что тестирование модуля завершено, и условия, при которых дальнейшее тестирование модуля считается невозможным до его изменения и доработки. Примером таких условий может служить достижение определенного уровня покрытия исходного кода тестами и невозможность компиляции модуля соответственно;

- определение требований к ресурсам - для разработки и выполнения тестов, а также для анализа результатов тестирования необходимы ресурсы - как технические (компьютеры и программное обеспечение), так и людские (тестировщики). При решении этой задачи необходимо указывать требования к программному и аппаратному обеспечению, требования к необходимой квалификации людей, а также должно определяться необходимое для проведения количество ресурсов и время их занятости;
- определение общего плана-графика работ - на основании общего плана проекта составляется план работ по модульному тестированию. Основной критерий начала работ по тестированию - готовность модулей, т.е. общий план работ по тестированию согласуется по датам начала работ с датами окончания работ общего плана разработки.

После завершения этапа планирования начинается этап определения свойств системы, подлежащих тестированию.

Основные задачи, которые решаются в ходе деятельности по определению свойств системы, подлежащих тестированию, включают в себя:

- изучение функциональных требований - определение тестопригодности требований, при необходимости запрашивается уточнение требований;
- определение дополнительных требований и связанных процедур - определение требований, которые не попадают под функциональные требования, но могут быть протестированы на уровне модульного тестирования (например, это могут быть требования к производительности системы, входящие в состав системных требований);
- определение состояний тестируемого модуля - если тестируемый модуль может быть представлен в виде конечного автомата с определенным набором состояний, то каждое состояние должно быть идентифицировано, а также должны быть выделены все требования, относящиеся к этому состоянию;
- определение характеристик входных и выходных данных - для всех данных, которые поступают в модуль, а также выходят из него, должны быть определены форматы, частота поступления, допустимые значения и т.п.;
- выбор элементов, подвергаемых тестированию - в случае, когда не может применяться полное тестирование, необходимо выбрать элементы тестируемого модуля, которые будут подвергаться тестированию. Основной источник информации здесь - данные о рисках, проанализированные на уровне структуры исходного кода тестируемого модуля. Для тестирования в первую очередь должны отбираться элементы с максимальной степенью риска.

И, наконец, в завершение фазы планирования производится уточнение основного плана - уточняется общий подход к тестированию, формулируются специальные и дополнительные требования к ресурсам, составляется детальный план-график работ.

По завершению этих этапов фаза планирования считается оконченной и начинается фаза разработки тестов. При этом процесс разработки тестов

подчиняется тем планам и требованиям, которые были созданы на предыдущем этапе. Таким образом, если на первом этапе основную роль выполнял руководитель группы тестирования, то на втором этапе основную роль начинает играть тестировщик, действующий в согласии с указаниями руководителя.

Фаза разработки тестов начинается с собственно разработки набора тестов, который будет использован для тестирования модуля. Основные документы, которые используются на этом этапе: функциональные требования к модулю, архитектура модуля, список элементов, подвергаемых тестированию, план-график работ, определения тестовых примеров от предыдущей версии модуля (если они существовали) и результаты тестирования прошлой версии (если они существовали).

В ходе этого этапа должны быть решены следующие задачи:

- разработка архитектуры тестового набора - под тестовым набором здесь понимается не набор конкретных тестовых примеров, а общая структура системы тестов для проверки функциональности тестируемого модуля. Организация тестов в такой системе как правило отражает структуру функциональных требований и зачастую представляет собой иерархию, на каждом уровне которой определяется свой набор тестов;
- разработка явных тестовых процедур (тест-требований) - в случае достаточно подробных функциональных требований и четко прописанной концепции разработки тестов явные тестовые процедуры могут и не разрабатываться. Однако, при наличии необходимых ресурсов, разработка тест-требований позволит более четко интерпретировать подвергаемые тестированию функциональные требования - тест-требования снижают риск неоднозначной трактовки функциональных требований;
- разработка тестовых примеров - тестовые примеры должны соответствовать требованиям к полноте тестирования и составляться либо на базе тест-требований, либо на основании функциональных требований. Данный вид деятельности наиболее продолжителен во времени;
- разработка тестовых примеров, основанных на архитектуре (в случае необходимости) - некоторые тестовые примеры основываются не на функциональных требованиях, а на особенностях архитектуры тестируемого модуля. Для разработки тестовых примеров, основанных на архитектуре, необходимо использовать подход стеклянного ящика. Эти тестовые примеры пишутся либо на основе низкоуровневых требований к системе, либо на основе низкоуровневых тест-требований, если они разрабатывались на одном из предыдущих этапов;
- составление спецификации тестовых примеров - результатом деятельности тестировщика в ходе данного этапа составляется документ Test Design Specification (формат которого описан в стандарте IEEE 829 [15]).

На следующем этапе проводится реализация тестов (например, в виде тестового окружения и формализованных описаний тестовых примеров). В ходе этого этапа формируются тестовые наборы данных, которые

используются в тестовых примерах, создается тестовое окружение, и также осуществляется интеграция тестового окружения с тестируемым модулем.

После того, как все тесты реализованы, они выполняются на тестовом стенде в ручном или автоматическом режиме. Вне зависимости от вида тестирования в ходе этого этапа решаются две задачи: выполнение тестовых примеров, и сбор и анализ результатов тестирования.

Сбору подлежит следующая информация:

- результат выполнения каждого тестового примера (прошел/не прошел);
- информация об информационном окружении системы в случае, если тест не прошел;
- информация о ресурсах, которые потребовались для выполнения тестового примера.

По результатам анализа этой информации составляются запросы на изменение проектной документации, программного кода тестируемого модуля или тестового окружения.

Этапы разработки (доработки), реализации и выполнения тестов продолжаются до тех пор, пока не будет достигнут критерий завершения модульного тестирования. Примером такого критерия может служить отсутствие не прошедших тестовых примеров при 75% покрытии строк исходного кода.

После прекращения тестирования выполняются работы по оценке проведенного тестирования, в ходе которых:

- описываются отличия реального процесса тестирования от запланированного;
- отличия поведения тестируемого модуля от описанного в требованиях (с целью дальнейшей коррекции требований);
- составляется общий отчет о прохождении тестов, включающий в себя и информацию о покрытии.

В завершение модульного тестирования необходимо проверить, что все созданные в его ходе артефакты - документы, программный код, файлы отчетов и данных - помещены в базу данных проекта, которая хранит все данные, используемые и создаваемые в процессе разработки программной системы.

JUnit5

Введение

JUnit — библиотека для модульного тестирования программ Java. Созданный Кентом Беком и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit для разных языков программирования, берущей начало в SUnit Кента Бека для Smalltalk. JUnit породил экосистему расширений — JMock, EasyMock, DbUnit, HttpUnit и т. д.

JUnit – это Java фреймворк для тестирования, т. е. тестирования отдельных участков кода, например, методов или классов.

Подключение зависимостей

Получить библиотеку JUnit5 можно несколькими способами, например:

- с помощью maven:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.1.0</version>
    <scope>test</scope>
</dependency>
```

- Или скачать JAR напрямую из репозитория maven: https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine

Пример простого теста

Приведен простейший пример использования JUnit5. Данный тест имеет всего одну проверку.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import example.util.Calculator;
import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

Аннотация @Test является маркером, указывающим на то, что данный метод является тестом.

Пример класса с тестами

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class StandardTests {
```

```
    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

Аннотации @BeforeAll и @AfterAll выполняют указанные методы перед и после запуска всех тестов в указанном классе. @BeforeEach и @AfterEach выполняют указанные методы перед и после запуска каждого теста в указанном классе.

Утверждения в тестах (Assertions)

Основные методы, определяющие вердикт тестирования:

- org.junit.jupiter.api.Assertions.assertEquals;
- org.junit.jupiter.api.Assertions.assertNotNull;
- org.junit.jupiter.api.Assertions.assertNull;
- org.junit.jupiter.api.Assertions.assertThrows;
- org.junit.jupiter.api.Assertions.assertTrue;
- org.junit.jupiter.api.Assertions.assertFalse;

Каждый из данных методов имеет множество перегруженных вариантов, которые подойдут почти на все необходимые случаи. Более подробно о них следует читать в официальной документации https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html

Допущения в тестах (Assumptions)

Допущения используются для запуска тестов только при соблюдении определенных условий. Обычно используется для внешних условий, которые необходимы для правильной работы теста, но которые не имеют прямого отношения к тому, что тестируется.

Вы можете объявить допущение с помощью assumeTrue(), assumeFalse(), assumingThat().

```
@Test
void trueAssumption() {
    assumeTrue(5 > 1);
    assertEquals(5 + 2, 7);
}

@Test
void falseAssumption() {
    assumeFalse(5 < 1);
    assertEquals(5 + 2, 7);
}

@Test
void assumptionThat() {
    String someString = "Just a string";
    assumingThat(
        someString.equals("Just a string"),
        () -> assertEquals(2 + 2, 4)
    );
}
```

Если предположение не выполняется, генерируется исключение TestAbortedException и тест просто пропускается.

Предположения также понимают лямбда-выражения.

Литература и ссылки

1. https://www.intuit.ru/studies/courses/1040/209/lecture/5409
2. https://junit.org/junit5/docs/current/user-guide/
3. https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html
4. https://www.baeldung.com/junit-5
5. https://www.baeldung.com/junit-5-preview

Вопросы для самоконтроля

1. Для чего нужно модульное тестирование?
2. Как создать модульный тест?
3. Какие основные методы утверждений вы помните (Assertions)?
4. Назовите основные аннотации, сопровождающие класс модульных тестов?